大O

一、时间复杂度

时间复杂度，渐进运行时间或者大O时间。

1.1 大O 、大θ、大Ω

学术界用大O、大θ、大Ω来描述运行时间。

- O。O描述时间的上界。
- Ω。描述时间的下界。表示没有比它更快的算法。
- θ。用θ同时表示O和Ω，即一个算法如果同时是O(N)且是Ω(N)的，他才是θ(N)的，它表示确界。

1.2 最优、最坏、期望

以快排为例。快排随机选择一个中点，通过数值交换把小于中点的放到前面、把大于中点的放到后面，然后使用相似流程递归排序左右两边的部分。

- 最优。如果所有元素相等，快排平均仅扫一次，也就是$O(N)$。
- 最坏。如果运气差，每次找到的元素都是剩余数组中最大的值。实际上，你选择的点是剩余数组的第一个元素，且剩余的数组是倒序排列的，就会遇到这种最坏的。这时候递归就失效了，每次仅仅是把数组少一个元素，就变成了$O(N^2)$。
- 期望情况。最优和最差的，通常不会发生，有时候中点可能很低或者很高，但不会一直如此，因此可以认为是$O(N\log(N))$。

1.3 最优、最坏、期望和大O、大Ω、大θ的关系

没有特别的关系。最优、最坏、期望是用来描述给定输入或者场景中的大O时间，大O、大Ω、大θ分别描述了运行时间的上界、下界、确界。

二、空间复杂度

时间复杂度不是算法唯一要关心的东西，还得关心内存数量和空间大小。

如果要创建大小为n的数组，需要的空间也是$O(n)$。若是创建n*n的二维数组，需要得空间为$O(n^2)$。

在递归的时候，栈空间也要算在内。如下面的代码，运行时间为$O(n)$,空间也是$O(n)$。

```
int sum(int n){
    if(n<=0>){
        return 0;
    }

    return n + sum(n - 1);
}
```

每次调用都会增加调用栈，这些调用中的每一次都会添加到调用栈中并占用实际的内存。

然而并不是调用n次就意味着需要$O(n)$的空间。看下面的函数：

```
int pairSumSequence(int n){
    int sum = 0 ;
    for (int i = 0 ; i < n ; i++){
        sum += pairSum(i , i + 1);
    }

    return sum;
}

int pairSum(int a , int b){
    return a + b;
}
```

pairSum大概调用n次，但是调用不等于同时发生，所以依然是需要$O(1)$的空间。

三、计算规则

3.1 常量

特定输入中，$O(N)$很有可能会比$O(1)$快，大O只是描述了增长的趋势。
因此常量不算在运行时间中，例如某个$O(2N)$的算法实际上是$O(N)$。

大O更多是表示运行时间的规模，我们需要知道：$O(N)$并不总是比$O(N^2)$快。

3.2 丢弃不重要的项

像$O(N^2+N)$这种表达式怎么计算？虽然第二个N不是常量，但是它跟N^2比起来，无关紧要。
所以应该舍弃无关紧要的数据。

- $O(N^2+N)$ -> $O(N^2)$
- $O(N+logN)$-> $O(N)$
- $O(5*2^N+1000*N^{100})$ -> $O(2^N)$

通常:

> $O(logx) < O(x) < O(xlogx) < O(x^2) < O(2^x) < O(X!)$

可以看出$O(X^2)$比$O(X)$糟糕很多，但是它比$O(2^x)$和$O(x!)$好很多。还有比$O(x!)$还差的，比如$O(x^x)$或者$O(2^x * x!)$。

四、多项式算法：加和乘

假如步骤有俩个，那么怎么区分应该是加还是乘呢？

- 如果步骤是先做A，再做B，则应该是加；
- 如果步骤是A步骤的每个步骤再做B步骤的每个步骤，应该是X。

举例如下:
加:

```
for(int a : arrA){
    print(a)
}

for(int b : arrB){
    print(b);
}
```

乘:

```
for(int a: arrA){
    for(int b: arrB){
        print(a + " ," + b);
    }
}
```

五、分摊时间

ArrayList或者动态数组，允许灵活改变大小。但是通常不会溢出，因为会动态扩容。
分摊时间：它描述了最坏的情况偶尔出现，一旦最坏的出现了，就会有很长一段时间不出现，也就是时间成本的分摊。

假设数组大小为2的幂数，插入一个元素的时候数组扩容2倍，所以当元素是X，那么以几何级数的扩容，每次加倍需要复制1，2，4，8，***，X个元素。所以换种思路计算，等于$X + X/2 + X/4 + ... + 1$最后约等于2X，因此X次插入需要$O(2X)$的时间，即每次插入的分摊时间约为$O(1)$.

六、LogN运行时间

典型如二分查找，从1开始每次乘以2，多少次到N？
$2^k = N$ => $k = logN$

还有平衡二叉树的查找。

七、递归运行时间

考虑下面的代码：

```
int f(int n){
    if(n<1){
        return 1;
    }

    return f(n-1)+f(n-1);
}
```

很显然，这不算是N^2。
假设调用f(4),则它调用f(3)两次，每个f(3)调用f(2)两次，直到f(1)。

所以:

> $2^0 + 2^1 + 2^2 + ... + 2^N = 2^{N+1}$

当一个多次调用自己的递归函数出现时，它的运行时间，往往是$O(分支数^{数的深度})$,分支数是每次调用自己的次数，所以上面的是2^N.